Storing and Querying XML Documents Without Using Schema Information

Kanda Runapongsa
Department of Computer Engineering
Khon Kaen University, Thailand
krunapon@kku.ac.th

Jignesh M. Patel
Department of EECS
University of Michigan, USA
jignesh@eecs.umich.edu

Abstract

As the popularity of eXtensible Markup Language (XML) continues to increase at an astonishing pace, data management systems for storing and querying large repositories of XML data are urgently needed. In this paper, we investigate using a Relational Database Management System (RDBMS) for storing and querying XML data. We present a mapping scheme, called PAID, for mapping XML documents to relations in an RDBMS. Compared to previously proposed mapping schemes, we demonstrate that the PAID mapping scheme results in less response times by up to several orders of magnitude. The primary reason for this performance improvement is that PAID includes information that the database can use to evaluate both direct and indirect containment queries efficiently.

Keywords: XML, RDBMS, XML Query Processing

1 Introduction

As the popularity of XML (eXtensible Markup Language) [4] for representing easily sharable data continues to grow, large repositories of XML data are likely to emerge. Data management systems for storing and querying these large repositories are urgently needed. Currently, there are two dominating approaches for managing XML repositories [8]. The first approach is to use a native XML database engine for storing and querying XML data sets [1, 14]. This approach has the advantage that it can provide a more natural data model and query language for XML data, which is typically viewed using a hierarchical or graph representation. The second approach is to map the XML data and queries to constructs provided by a Relational DBMS (RDBMS) [5, 9, 18, 19]. XML data is mapped to relations, and queries on the XML data are converted into SQL queries. The results of the SQL queries are then converted to XML documents before returning the answer to the user. Using an RDBMS, one can leverage many decades of research and commercialization efforts by exploiting existing features in such database. An additional advantage of an RDBMS is that it can be used for querying both XML data and data that exists in the relational systems. In this paper, we focus on using an RDBMS as an experimental platform to store and query XML documents.

It is necessary to find an efficient method for processing XML documents without using schema information because many XML documents exist without associated schemas. A number of different approaches have been proposed to map XML data to relations, independent of XML schemas [9, 19–21]. However, no approach has been shown to be consistently superior to other methods. In this paper, we propose a mapping scheme which outperforms other mapping schemes.

To efficiently evaluate XML queries, it is important to quickly determine the structural relationships among any pair of nodes. Structural relationships can be determined by associating each node with a numbering scheme. One of the most popular method is Dietz's numbering scheme [6], which has been adopted by several XML query evaluation techniques [2, 10, 13, 21]. In this numbering scheme, a document is modeled as a tree. Each tree node is assigned three numbers: the node's *preorder* and *postorder* ranks, and the node's level. In a preorder (postorder) traversal, a node is visited and assigned its preorder (postorder) rank before (after) its children are recursively assigned from left to right. If the document is read sequentially, elements are accessed in the same order as their preorder ranks. Using Dietz's numbering scheme, the following properties hold:

- Node v is an ancestor of node u iff (i) $preorder(v) < preorder(u)$ and (ii) $postorder(u) < postorder(v)$. In other words, node u is indirectly contained in node v.

- Node v is a parent of node u iff (i) $preorder(v) < preorder(u)$, (ii) $postorder(u) < postorder(v)$, and (iii) $level(v) = level(u) - 1$. In other words, node u is directly contained in node v.

To evaluate path expressions in a relational system, there are two dominating approaches. The first is to use primary-foreign keys of relations to establish a parent-child relationship between elements [9,18]. The other approach is to associate each element with a numbering scheme (preorder, postorder, level) [19–21]. The advantage of the primary-foreign keys approach is that the queries containing only parent-child relationships are evaluated very quickly since there are indices built on the primary-foreign keys. However, the disadvantage of this approach is the poor performance of ancestor-descendant queries. When using the primary-foreign keys approach to evaluate ancestor-descendant queries, the number of join operations can be as many as the number of steps of the longest path in the document tree [20]. On the other hand, using node positions, only direct joins between ancestors and descendants are needed. Another problem of the primary-foreign keys approach is that ancestor-descendant queries are very expensive to evaluate [11].

In this paper, we focus on evaluating XML queries using the node positions approach because of its many disadvantage and it does not require schema information. There have been several works that use node positions in their mapping schemes. We compare well-known mapping schemes with our proposed mapping scheme. We improve existing mapping schemes by encoding extra information about the node's parent and including the node's path information. Our experimental results show that this additional information can dramatically reduce query response times in a relational system. Although we did not perform the experiments on a native XML database, the techniques presented in this paper can be applied in such database.

The remainder of this paper is organized as follows. We first discuss different mapping approaches in Section 2. We then evaluate and compare the effectiveness of these mapping approaches in Section 3. Related work is discussed in Section 4. Finally, we present our conclusions and discuss future work in Section 5.

2 Different Mapping Approaches

In this section, we present three different mapping approaches: two other mapping approaches and one our proposed mapping approach. All of these approaches model a document as a tree and use the word positions of elements to establish the structural relationships between elements, but they include information about node positions and node paths differently. We use the sample XML document in Figure 1 as an input document to these approaches and then present the mapped relations for each of them. The input document describes a play consisting a list of speakers with their speeches.

We first present the relational schemas and the sample mapped relations of the approach which we refer to as Begin-End-Level (BEL) [21] and those of the approach which we refer to as Begin-End-Level-Path (BELP) [19]. We then present our proposed mapping approach which we refer to as Parent-ID (PAID).

2.1 The Begin-End-Level (BEL) Approach

In the Begin-End-Level (BEL) approach [21], containment queries are processed using inverted lists. Each inverted list records the occurrences of a word or an element which is referred to as "`term`". Each occurrence is indexed by its document number (`docno`), its word position (`begin`:`end`), and its nesting depth within the document (`level`). The positions of elements and words (`begin`, `end`, `wordno`) are generated by counting word numbers in an input XML document. The occurrence of an element is denoted as (`docno`, `begin`:`end`, `level`), and the occurrence of a text word is denoted as (`docno`, `wordno`, `level`). The `begin` position of the node represents the preorder of node, and the `end` position of the node represents the postorder of the node. The `wordno` represents both the preorder and the postorder of the text word. Using these `begin`, `end`, `wordno`, one can determine the structural relationship between nodes in XML documents. Elements and text are stored in the `element` and `text` relations, respectively. The storage of attributes is not discussed in [21].

To handle documents that have attributes, one can extend this approach by treating an attribute like an element and append attribute information in `element`. However, from our experimental results, separately storing attribute

```
<?xml version="1.0"?>
1
<PLAY>
  2
  <ACT>
          3
          <TITLE>Act One</TITLE>
          7
          <SCENE>
                  8
                  <TITLE>Scene One</TITLE>
                  12
                  <SPEECH>
                          13
                          <SPEAKER>Romeo</SPEAKER>
                          16
                          <LINE>Hello, Juliet</LINE>
                  20
                  </SPEECH>
                  21
                  <SPEECH>
                          22
                          <SPEAKER>Juliet</SPEAKER>
                          25
                          <LINE>Good morning, Romeo</LINE>
                  30
                  </SPEECH>
          31
          </SCENE>
  32
  </ACT>
  33
</PLAY>
```

Figure 1: A Sample XML Document (with Word Positions in Bold)

information in the `attribute` relation results in superior performance. To answer the queries with order predicates, we add the `order` attribute in the `element` relation. The schemas of the three relations are as follows:

- `element(term string, docID integer, order integer, begin integer, end integer, level integer)`.
- `text(term string, docID integer, wordno integer, level integer)`.
- `attribute(term string, value string, docID integer, begin integer, end integer, level integer)`.

Using this approach, the content of the relations storing sample XML data in Figure 1 is shown in Figures 2-3. Note that there is no `attribute` relation illustrated here because the input sample XML document does not have any attribute. Suppose that one wants to retrieve all speakers, the path expression for this request is SPEAKER. The corresponding SQL query for the path expression SPEAKER is shown in Figure 4.

2.2 The Begin-End-Level-Path (BELP) Approach

In [19, 20], the Begin-End-Level-Path (BELP) approach decomposes an XML document into nodes on the basis of its tree structures. A relation is created for each node type, and a path relation stores path information from the root node to each of other nodes. Nodes of different XML documents are stored in the same relation as long as they are of the same type.

In the original BELP approach [19], the position of the node has the REGION type (User Abstract Data Type) [19]. We do not store the position within the REGION type because our experimental results indicate that it is more effective

term	docID	order	begin	end	level
PLAY	1	1	1	33	1
ACT	1	1	2	32	2
TITLE	1	1	3	6	3
SCENE	1	1	7	31	3
TITLE	1	1	8	11	4
SPEECH	1	1	12	20	4
SPEAKER	1	1	13	15	5
LINE	1	1	16	19	5
SPEAKER	1	1	22	24	5
SPEECH	1	2	21	30	4
LINE	1	1	25	29	5

Figure 2: The `element` Relation of BEL

term	docID	wordno	level
Act	1	4	4
One	1	5	4
Scene	1	9	5
One	1	10	5
Romeo	1	14	6
Hello	1	17	6
Juliet	1	18	6
Juliet	1	23	6
Good	1	26	6
morning	1	27	6
Romeo	1	28	6

Figure 3: The `text` Relation of BEL

```
select  tspeaker.term
from    element espeaker,   text tspeaker
where   espeaker.term = ‘SPEAKER’
and     espeaker.begin    <   tspeaker.begin
and     tspeaker.end      <   espeaker.end
and     espeaker.level    =   tspeaker.level - 1
and     espeaker.docID    =   tspeaker.docID
```

Figure 4: A SQL Query When Using BEL

to store the begin and end positions as attributes on which the B+ tree indices can be built. Traditional RDBMSs do not allow the users to build indices on a user-defined type, such as REGION. Although the level information is not originally included [19], we include it to distinguish between the parent-child and the ancestor-descendant relationships.

In the BELP approach, each word occurrence is assigned an integer corresponding to its position within the document, and each tag is assigned a real number. The integer part of the real number indicates the position number of the preceding word. The decimal part indicates the position of the tag in the current depth-first traversal. When visiting the root node or the node visited immediately following the text word, the decimal part is initialized to one. When visiting the other nodes, the decimal part is incremented. The position of the element is assigned a real number to minimize the effects of the appearances of the element tags on a word-based proximity search on element content [19]. Figure 5 shows the data graph with the assignment of the region of each node in the XML data of Figure 1.

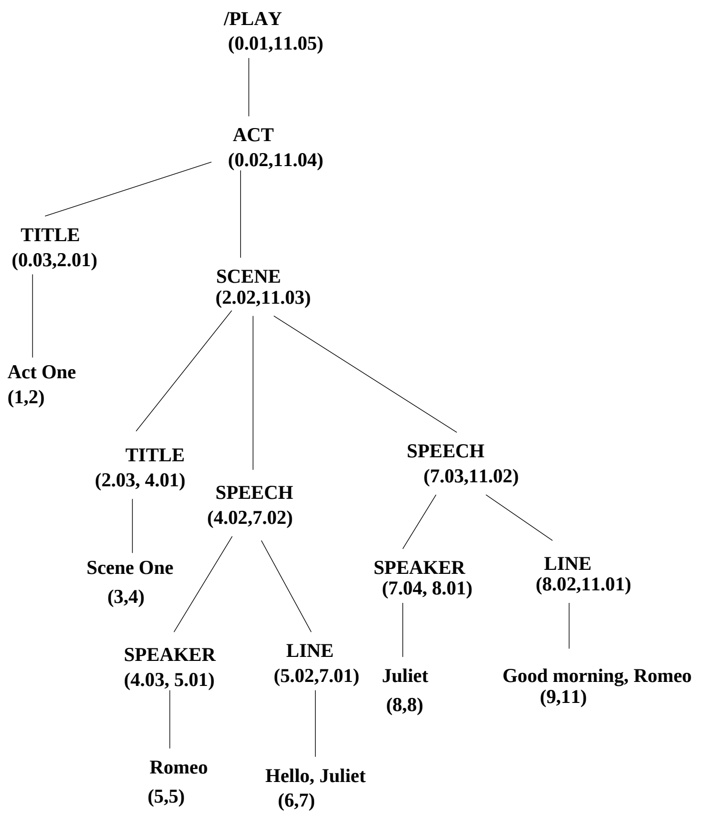

Figure 5: The Tree Structure in the Sample Document When Using BELP

The details of the schemas of the `element`, `attribute`, `text`, and `path` relations are as follows:

- `element(docID integer, pathID integer, order integer, begin double, end double, level integer)`.
- `attribute(docID integer, pathID integer, attvalue string, begin double, end double, level integer)`.
- `text(docID integer, pathID integer, value string, begin double, end double, level integer)`.
- `path(pathExp string, pathID integer)`.

Using this approach, the content of the relations storing sample XML data in Figure 1 is shown in Figures 6-8. The corresponding SQL query for the path expression `SPEAKER` is shown in Figure 9.

docID	pathID	order	begin	end	level
1	0	1	0.01	11.05	1
1	1	1	0.02	11.04	2
1	2	1	0.03	2.01	3
1	3	1	2.02	11.03	3
1	2	1	2.03	4.01	4
1	5	1	4.02	7.02	4
1	6	1	4.03	5.01	5
1	7	1	5.02	7.01	5
1	5	2	7.03	11.02	4
1	6	1	7.04	8.01	5
1	7	1	8.02	11.01	5

Figure 6: The `element` Relation of BELP

docID	pathID	value	begin	end	level
1	2	Act One	1	2	4
1	4	Scene One	3	4	5
1	6	Romeo	5	5	6
1	7	Hello, Juliet	6	7	6
1	6	Juliet	8	8	6
1	7	Good morning, Romeo	9	11	6

Figure 7: The `text` Relation of BELP

pathExp	pathID
/PLAY	0
/PLAY/ACT	1
/PLAY/ACT/TITLE	2
/PLAY/ACT/SCENE	3
/PLAY/ACT/SCENE/TITLE	4
/PLAY/ACT/SCENE/SPEECH	5
/PLAY/ACT/SCENE/SPEECH/SPEAKER	6
/PLAY/ACT/SCENE/SPEECH/LINE	7

Figure 8: The `path` Relation of BELP

```
select  tspeaker.value
from    element tspeaker,   path pspeaker,
        text tspeaker,      path tspeaker
where   espeaker.pathExp    like  '%SPEAKER'
and     tspeaker.pathExp    like  '%SPEAKER'
and     tspeaker.pathID     =     espeaker.pathID
and     espeaker.begin      <     tspeaker.begin
and     tspeaker.end        <     espeaker.end
and     espeaker.level      =     tspeaker.level - 1
and     espeaker.docID      =     tspeaker.docID
```

Figure 9: A SQL Query When Using BELP

2.3 The Parent-ID (PAID) Approach

As other approaches, the Parent-ID (PAID) approach uses (being, end, level) for determining containment relationship queries. However, unlike other mapping schemes, the PAID approach includes both the IDs of the parent nodes to quickly determine elements that have parent-child relationships as well as the path information to quickly retrieve elements that match a given path. The schemas of the relations in this approach are as follows:

- `element(docID integer, term integer, pathID integer, order integer, begin integer, end integer, level integer, parentID integer)`.
- `path(pathExp string, pathID integer)`.
- `attribute(attrName string, docID integer, begin integer, attrValue string, level integer, parentID integer)`.
- `text(term string, docID integer, wordno integer, level integer, parentID integer)`.

Using this approach, the content of the relations storing the sample XML data shown in Figure 1 is depicted in Figures 10-12. With the attribute `parentID`, one can evaluate path expression `SPEAKER` using a SQL statement as shown in Figure 13.

docID	term	pathID	order	begin	end	level	parentID
1	TITLE	2	1	3	6	3	2
1	TITLE	4	1	8	11	4	7
1	SPEAKER	6	1	13	15	5	12
1	LINE	7	1	16	19	5	12
1	SPEECH	5	1	12	20	4	7
1	SPEAKER	6	1	22	24	5	21
1	LINE	7	1	25	29	5	21
1	SPEECH	5	2	21	30	4	7
1	SCENE	3	1	7	31	3	2
1	ACT	1	1	2	32	2	1
1	PLAY	0	1	1	33	1	-1

Figure 10: The `element` Relation of PAID

pathExp	pathID
/PLAY	0
/PLAY/ACT	1
/PLAY/ACT/TITLE	2
/PLAY/ACT/SCENE	3
/PLAY/ACT/SCENE/TITLE	4
/PLAY/ACT/SCENE/SPEECH	5
/PLAY/ACT/SCENE/SPEECH/SPEAKER	6
/PLAY/ACT/SCENE/SPEECH/LINE	7

Figure 11: The `path` Relation of PAID

3 Performance Evaluation

In this section, we present the performance of different mapping approaches. We evaluated the effectiveness of the three mapping approaches using the well-known Shakespeare plays data set [3].

term	docID	wordno	level	parentID
Act	1	4	4	3
One	1	5	4	3
Scene	1	9	5	8
One	1	10	5	8
Romeo	1	14	6	13
Hello	1	17	6	16
Juliet	1	18	6	16
Juliet	1	23	6	22
Good	1	26	6	25
morning	1	27	6	25
Romeo	1	28	6	25

Figure 12: The `text` Relation of PAID

```
select  tspeaker.term
from    element espeaker,   text tspeaker
where   espeaker.term = ‘SPEAKER’
and     espeaker.begin     =     tspeaker.parentID
and     espeaker.docID     =     tspeaker.docID
```

Figure 13: A SQL Query When Using PAID

3.1 Experimental Setup

We used the Apache Xerces C++ version 2.0 [17] to parse the documents and generate the content of relations in different mapping approaches.

We performed all experiments using a leading commercial RDBMS on a single-processor 1.2 GHz Pentium Celeron machine running Windows XP with 256 MB of main memory. The buffer pool size of the database was set to 32 MB.

Before executing queries in any approach, we collected statistics and created indices as suggested by the index selection tool of the database. The execution times reported in this section are cold numbers. Each query was run five times, and the average of the middle three execution times was reported.

In the experiment, we loaded the Shakespeare play documents into the database using the three mapping approaches. To have the large size of the experimental data, we used eight copies of the original Shakespeare data set which has approximately 8 MB. Thus, the total input data size is 64 MB.

Table 1 shows the comparisons of the database and index sizes of different mapping approaches. The PAID approach has the largest database size since each relation has additional attributes, such as the `parentID` attribute. The large index size of the BEL approach is due to the large size of indices created on the `value` attribute of the `text` relation. The BELP approach has a smaller total index size because there is no index created on the `value` attribute. The length of the `value` attribute is the same as the maximum length of the element content which can be very long. The database does not create the index on such long string attribute because it is inefficient to create and search for objects using a large key.

	BEL	BELP	PAID
Database size (MB)	198	151	231
Index size (MB)	505	189	360

Table 1: Database and Index Sizes for the Shakespeare Data Set

The XPath expressions of the query workload are described in Figure 14.

The result of executing queries for the Shakespeare data set are shown in Table 2.

Figure 15 is a graphical representation to further emphasize the performance difference of these approaches. This

```
QS1. ACT/SCENE/SPEECH/LINE[contains(STAGEDIR, ‘Rising’)]
QS2. ACT/SCENE/SPEECH/LINE[STAGEDIR]
QS3. ACT/SCENE/SPEECH/SPEAKER, ACT/SCENE/SPEECH/LINE
QS4. /PLAY[contains(TITLE,‘Juliet’)]/ACT/SCENE/
     SPEECH[contains(SPEAKER,‘ROMEO’)]
QS5. /PLAY[contains(TITLE,‘Juliet’)]//ACT/SCENE/
     SPEECH[contains(LINE,‘love’)][contains(SPEAKER,‘ROMEO’)]
QS6. PROLOGUE/SPEECH/LINE[position()=2]
```

Figure 14: The Query Workload on the Shakespeare Data Set

Query	Execution Times (seconds)		
	BEL	BELP	PAID
QS1	24.92	30.70	0.03
QS2	81.39	18.46	10.29
QS3	367.39	836.00	30.99
QS4	10.67	23.35	1.42
QS5	580.40	952.09	56.61
QS6	3.54	0.11	0.01

Table 2: Execution Times of the Three Mapping Approaches for the Shakespeare Data Set

figure illustrates the ratios of the execution times of other approaches over the PAID approach on a **log** scale.

As shown in Figure 15, we observe that the PAID approach outperforms other approaches by several orders of magnitude, especially queries QS1. The response times of queries using the PAID approach are much less than those using other approaches because of three primary factors: 1) PAID uses the parentID attribute to quickly find the parent nodes, 2) PAID uses the path information to reduce the number of join operations in long path queries, and 3) PAID uses the index on the `value` attribute to quickly retrieve the nodes that satisfy with the value predicates.

Examining the query execution times shown in Table 2, compared to BELP, BEL performs better on queries with value predicates (QS1, QS4, and QS5) because there is the index on the `value` attribute of the `text` relation of the BEL approach. In BELP, there is no index on such attribute because of the long length of the text value. BEL also outperforms BELP for queries with multiple path expressions (QS3, QS4, and QS5) because, unlike BELP, BEL does not require sorting. In BELP, elements are first searched by the specified path expression. Then, the elements that match the given paths are combined with a sort merge join operation in which the nodes are sorted by their positions. On the other hand, in BEL, the inputs of the join operations are attributes that are retrieved from clustered indices. Thus, there is no need for sorting when performing the join between these attributes.

4 Related Work

McHugh et al. [15] described techniques for building and exploiting indices in Lore [14]. Their path index always begins at the root node and is not applicable to regular path expressions. Milo and Suciu [16] proposed a template index (T-index) as a general index structure for semistructured databases. The T-index allows users to sacrifice space for generality and to query a wide range of regular path expressions. However, the index generation tends to be complex. Although the authors reported the sizes of the T-index for different data sets, they did not report any query response time numbers.

A few recently proposed indexing structures [12, 13] have addressed numbering schemes that accommodate updates of the source data. Kha et al. [12] proposed an indexing structure scheme based on the Relative Region Coordinate (RRC) in which the coordinate of an XML element is based on the region of its parent element. Li and Moon [13] have developed a new XML Indexing and Storage System (XISS) in which the node is assigned a pair of numbers $<$order, size$>$. For a tree node y and its parent x, $order(x) < order(y)$ and $order(y)+size(y) \leq order(x)+size(x)$. A disadvantage of this numbering scheme is that it is difficult to estimate a size that can accommodate an arbitrarily

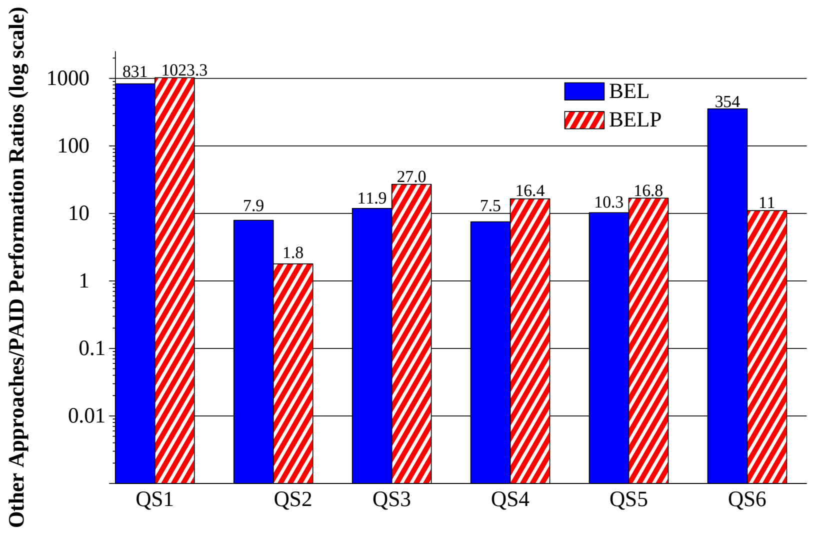

Figure 15: Other Approaches/PAID Performance Ratios on a Log Scale for the Shakespeare Data Set

large number of insertions.

Some proposed indexing structures are implemented and experimented on existing DBMSs [7,19–21]. Fegaras and Elmasri [7] presented an inverse indexing technique that the indices on content words and elements in XML documents are stored in an Object-Oriented Database Management System (OODBMS). Shimura et al [19] decomposed the tree structure of XML documents into nodes and stored them in relational tables according to their types. The advantage of their approach is that it is independent of XML schemas. Later, Zhung et al. [21] transformed the inverted index into relational tables and converted containment queries into SQL queries. Their performance study shows that efficient join algorithms and hardware cache utilization can significantly improve the performance of an RDBMS in supporting containment queries. Unlike these approaches, we add the node's parent information and the node's path information to the numbering scheme (preorder, postorder, level) so that both the parent-child and ancestor-descendant relationships between elements (both the direct and indirect containment queries) can be determined quickly.

5 Conclusions

We have performed a study evaluating different mapping approaches for storing and querying XML data in an RDBMS without using schemas of XML documents. We demonstrate that the path information and the parent node information can significantly improve the performance of query evaluation. The performance of evaluating a query is very sensitive to the SQL rewrite. Some of the insights from our experimental results suggest the following techniques:

- Using the path information to identify elements that match the path for a query consisting of a single path expression.
- Using the element tag name when a query consisting of multiple path expressions.
- Storing multiple words of each element content together when a query has many multi-word predicates.
- Separately storing each word of the element content when a query is dominated by single-word predicates.
- Using the parent ID information when joining elements that are selective.

For the future work, we are interested in taking the query workload and data statistics into account when determining a mapping scheme. Another extension is to have a mapping such that elements that appear several times are stored only once in relations so that the storage space and the maintenance time are reduced.

References

[1] Software AG. Tamino - The Information Server for Electronic Business, 2000. `http://www.softwareag.com/tamino/technical/wp_download.htm`.

[2] S. Al-Khalifa, H.V.Jagadish, N.Koudas, J.M.Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the IEEE International Conference on Data Engineering*, San Jose, CA, February 2002.

[3] J. Bosak. The Plays of Shakespeare in XML, July 1999. `http://metalab.unc.edu/xml/examples/shakespeare/`.

[4] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler (eds). Extensible Markup Language (XML), October 2000. `http://www.w3.org/TR/REC-xml`.

[5] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 431–442. ACM Press, 1999.

[6] P. F. Dietz. Maintaining Order in a Linked List. In *Proceedings of the Fourtheenth Annual ACM Symposium of Theory of Computing*, pages 122–127, San Francisco, California, May 1982.

[7] L. Fegaras and R. Elmasri. Query Engines for Web-Accessible XML Data. In *Proceedings of the International Conference on Very Large Data Bases*, Rome, Italy, September 2001.

[8] D. Florescu, G. Graefe, G. Moerkotte, H. Pirahesh, and H. Schning. Panel: XML Data Management: Go Native or Spruce up Relational Systems? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, May 2001.

[9] D. Flosescu and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. In *Rapport de Recherche No.3684*, INRIA,Rocquencourt,France, March 1999.

[10] T. Grust. Accelerating XPath Location Steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, 2002.

[11] K. Runapongsa and J. M. Patel and H.V. Jagadish and S. Al-Khalifa. The Michigan Benchmark:A Microbenchmark for XML Query Processing Systems. In *EEXTT*, pages 160–161, Hong Kong, China, August 2002.

[12] D. D. Kha, M. Yoshikawa, and S. Uemura. An XML Indexing Structure with Relative Region Coordinate. In *ICDE*, pages 313–320, Heidelberg, Germany, April 2001.

[13] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 361–370, September 2001.

[14] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, September 1997.

[15] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran. Indexing Semistructured Data. Technical report, Computer Science Department, Stanford University, 1998. `http://www-db.stanford.edu/lore/pubs/semiindexing98.pdf`.

[16] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of the International Conference on Database Teorey*, pages 277–295, Jerusalem, Israel, January 1999.

[17] The Apache XML Project. Xerces C++ Parser. `http://xml.apache.org/xerces-c/index.html`.

[18] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D.DeWitt, and J.Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the International Conference on Very Large Data Bases*, pages 302–314, Edinburgh, Scotland, September 1999.

[19] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *International Conference on Database and Expert Systems Applications*, pages 206–217, Florence, Italy, September 1999.

[20] M. Yoshikawa, T. Amagasa, T. Shimura, and S. S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, 1(1):110–141, August 2001.

[21] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Managment Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, May 2001.